Attorney's Docket No.: 10559-615001 / P12854

APPLICATION

FOR

UNITED STATES LETTERS PATENT

PATENT APPLICATION

TITLE: FUNCTIONAL PIPELINES

APPLICANT: HUGH M. WILKINSON III, MATTHEW J. ADILETTA, GILBERT WOLRICH, MARK B. ROSENBLUTH, DEBRA BERNSTEIN AND MYLES J. WILDE

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL870691335US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

1-17-02

Date of Deposit

Signature

Gabriel Lewis

Typed or Printed Name of Person Signing Certificate

FUNCTIONAL PIPELINES

BACKGROUND

This invention relates to functional pipelines.

Parallel processing is an efficient form of information processing of concurrent events of a computing system. Parallel processing demands concurrent execution of many programs, in contrast to sequential processing. In the context of parallel processing, parallelism involves doing more than one thing at the same time. Unlike a serial paradigm where all tasks are performed sequentially at a single station or a pipelined machine where tasks are performed at specialized stations, with parallel processing, many stations are provided, each capable of performing and carrying out various tasks and functions simultaneously. A number of stations work simultaneously and independently on the same or common elements of a computing task. Accordingly, parallel processing solves various types of computing tasks and certain problems are suitable for solution by applying several instruction processing units and several data streams.

Express Mail Label No. EV339054505US

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a processing system.

FIG. 2 is a detailed block diagram of the processing system of FIG. 1.

FIG. 3 is a block diagram of a programming engine of the processing system of FIG. 1.

FIG. 4 is a block diagram of a functional pipeline unit of the processing system of FIG. 1.

FIG. 5 is a block diagram illustrating details of the processing system of FIG. 1.

DESCRIPTION

Architecture:

Referring to FIG. 1, a computer processing system 10 includes a parallel, hardware-based multithreaded network processor 12. The hardware-based multithreaded processor 12 is coupled to a memory system or memory resource 14. Memory system 14 includes dynamic random access memory (DRAM) 14a and static random access memory 14b (SRAM). The processing system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically, the hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple functional microengines or programming engines 16 each with multiple hardware controlled

threads that are simultaneously active and independently work on a specific task.

The programming engines 16 each maintain program counters in hardware and states associated with the program counters. Effectively, corresponding sets of context or threads can be simultaneously active on each of the programming engines 16 while only one is actually operating at any one time.

In this example, eight programming engines 16a-16h are illustrated in FIG. 1. Each engine from the programming engines 16a-16h processes eight hardware threads or contexts. The eight programming engines 16a-16h operate with shared resources including memory resource 14 and bus interfaces (not shown). The hardware-based multithreaded processor 12 includes a dynamic random access memory (DRAM) controller 18a and a static random access memory (SRAM) controller 18b. The DRAM memory 14a and DRAM controller 18a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM memory 14b and SRAM controller 18b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and the like.

The eight programming engines 16a-16h access either the DRAM memory 14a or SRAM memory 14b based on characteristics of the

data. Thus, low latency, low bandwidth data is stored in and fetched from SRAM memory 14b, whereas higher bandwidth data for which latency is not as important, is stored in and fetched from DRAM memory 14a. The programming engines 16a-16h can execute memory reference instructions to either the DRAM controller 18a or SRAM controller 18b.

The hardware-based multithreaded processor 12 also includes a processor core 20 for loading microcode control for the programming engines 16a-16h. In this example, the processor core 20 is an XScale™ based architecture.

The processor core 20 performs general purpose computer type functions such as handling protocols, exceptions, and extra support for packet processing where the programming engines 16 pass the packets off for more detailed processing such as in boundary conditions.

The processor core 20 has an operating system (not shown). Through the operating system (OS), the processor core 20 can call functions to operate on the programming engines 16a-16h. The processor core 20 can use any supported OS, in particular, a real time OS. For the core processor 20 implemented as an XScale™ architecture, operating systems such as Microsoft NT real-time, VXWorks and µCOS, or a freeware OS available over the Internet can be used.

Advantages of hardware multithreading can be explained by SRAM or DRAM memory accesses. As an example, an SRAM access requested by a context (e.g., Thread_0), from one of the programming engines 16 will cause the SRAM controller 18b to initiate an access to the SRAM memory 14b. The SRAM controller 18b accesses the SRAM memory 14b, fetches the data from the SRAM memory 14b, and returns data to a requesting programming engine 16.

During an SRAM access, if one of the programming engines 16a-16h had only a single thread that could operate, that programming engine would be dormant until data was returned from the SRAM memory 14b.

By employing hardware context swapping within each of the programming engines 16a-16h, the hardware context swapping enables other contexts with unique program counters to execute in that same programming engine. Thus, another thread e.g., Thread_1 can function while the first thread, Thread_0, is awaiting the read data to return. During execution, Thread_1 may access the DRAM memory 14a. While Thread_1 operates on the DRAM unit, and Thread_0 is operating on the SRAM unit, a new thread, e.g., Thread_2 can now operate in the programming engine 16. Thread_2 can operate for a certain amount of time until it needs to access memory or perform some other long latency operation,

such as making an access to a bus interface. Therefore, simultaneously, the multi-threaded processor 12 can have a bus operation, an SRAM operation, and a DRAM operation all being completed or operated upon by one of the programming engines 16 and have one more threads or contexts available to process more work.

The hardware context swapping also synchronizes the completion of tasks. For example, two threads can access the shared memory resource, e.g., the SRAM memory 14b. Each one of the separate functional units, e.g., the SRAM controller 18b, and the DRAM controller 18a, when they complete a requested task from one of the programming engine threads or contexts reports back a flag signaling completion of an operation. When the programming engines 16a-16h receive the flag, the programming engines 16a-16h can determine which thread to turn on.

One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a Media Access Controller (MAC) device, e.g., a 10/100BaseT Octal MAC 13a or a Gigabit Ethernet device 13b. In general, as a network processor, the hardware-based multithreaded processor 12 can interface to any type of communication device or interface that receives or sends

large amount of data. The computer processing system 10 functioning in a networking application can receive network packets and process those packets in a parallel manner.

Programming Engine Contexts:

Referring to FIG. 2, one exemplary programming engine 16a from the programming engines 16a-16h, is shown. The programming engine 16a includes a control store 30, which in one example includes a RAM of 4096 instructions, each of which is 40-bits wide. The RAM stores a microprogram that the programming engine 16a executes. The microprogram in the control store 30 is loadable by the processor core 20 (FIG. 1).

In addition to event signals that are local to an executing thread, the programming engine 16a employs signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all programming engines 16a-16h. Any and all threads in the programming engines can branch on these signaling states. These signaling states can be used to determine availability of a resource or whether a resource is due for servicing. The context event logic has arbitration for the eight (8) threads. In one example, the arbitration is a round robin mechanism. Other techniques could be used including priority queuing or weighted fair queuing.

As described above, the programming engine 16a supports multi-threaded execution of eight contexts. This allows one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. Multi-threaded execution is critical to maintaining efficient hardware execution of the programming engine 16a because memory latency is significant. Multi-threaded execution allows the programming engines 16 to hide memory latency by performing useful independent work across several threads.

The programming engine 16a, to allow for efficient context swapping, has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context specific information to and from shared memory and programming engine registers for each context swap. Fast context swapping allows a context to do computation while other contexts wait for input-output (I/O), typically, external memory accesses to complete or for a signal from another context or hardware unit.

For example, the programming engine 16a executes the eight contexts by maintaining eight program counters and eight context relative sets of registers. There can be six different types of context relative registers, namely, general purpose registers

(GPRs) 32, inter-programming agent registers (not shown), Static Random Access Memory (SRAM) input transfer registers 34, Dynamic Random Access Memory (DRAM) input transfer registers 36, SRAM output transfer registers 38, DRAM output transfer registers 40.

The GPRs 32 are used for general programming purposes. The GPRs 32 are read and written exclusively under program control. The GPRs 32, when used as a source in an instruction, supply operands to an execution datapath 44. When used as a destination in an instruction, the GPRs 32 are written with the result of the execution datapath 44. The programming engine 16a also includes I/O transfer registers 34, 36, 38 and 40 which are used for transferring data to and from the programming engine 16a and locations external to the programming engines 16a, e.g., the DRAM memory 14a, the SRAM memory 14b, and etc.

A local memory 42 is also used. The local memory 42 is addressable storage located in the programming engine 16a. The local memory 42 is read and written exclusively under program control. The local memory 42 also includes variables shared by all the programming engines 16a-16h. Shared variables are modified in various assigned tasks during functional pipeline stages by the programming engines 16a-16h, which are described next. The shared variables include a critical section, defining the read-modify-write times. The implementation and use of the

critical section in the computing processing system 10 is also described below.

Functional Pipelining and Pipeline Stages:

Referring to FIG. 3, the programming engine 16a is shown in a functional pipeline unit 50. The functional pipeline unit 50 includes the programming engine 16a and a data unit 52 that includes data, operated on by the programming engine, e.g., network packets 54. The programming engine 16a is shown having a local register unit 56. The local register unit 56 stores information from the data packets 54.

In the functional pipeline unit 50, the contexts 58 of the programming engines 16a, namely, Programming Engine0.1 (PE0.1) through Programming Engine0.n (PE0.n), remain with the programming engine 16a while different functions are performed on the data packets 54 as time 66 progresses from time = 0 to time = t. A programming execution time is divided into "m" functional pipeline stages or pipe-stages 60a-60m. Each pipeline stage of the pipeline stages 60a-60m performs different pipeline functions 62a, 64, or 62p on data in the pipeline.

The pipeline stage 60a is, for example, a regular time interval within which a particular processing function, e.g., the function 62 is applied to one of the data packets 54. A

processing function 62 can last one or more pipelines stages 62. The function 64, for example, lasts two pipeline stages, namely pipeline stages 60b and 60c.

A single programming engine such as the programming engine 16a can constitute a functional pipeline unit 50. In the functional pipeline unit 50, the functions 62a, 64, and 62p move through the functional pipeline unit 50 from one programming engine 16 to another programming engine 16, as will be described next.

Referring to FIG. 4, the data packets 54 are assigned to programming engine contexts 58 in order. Thus, if "n" threads or contexts 58 execute in the programming engine 16a, the first context 58, "PE0.1" completes processing of the data packet 54 before the data packets 54 from the "PE0.n" context arrives. With this approach the programming engine 16b can begin processing the "n+1" packet.

Dividing the execution time of the programming engines 16 into functional pipeline stages 60a-60c results in more than one programming engine 16 executing an equivalent functional pipeline unit 70 in parallel. The functional pipeline stage 60a is distributed across two programming engines 16a and 16b, with each of the programming engines 16a and 16b executing eight contexts each.

In operation, each of the data packets 54 remains with one of the contexts 58 for a longer period of time as more programming engines 16 are added to the functional pipeline units 50 and 70. In this example, the data packet 54 remains with a context sixteen data packet arrival times (8 contexts x 2 programming engines) because context PE0.1 is not required to accept another data packet 58 until the other contexts 58 have received their data packets.

In this example, the function 62 of the functional pipeline stage 60a can be passed from the programming engine 16a to the programming engine 16b. Passing of the function 62 is accomplished by passing the processing functions from one programming engine to another, as illustrated by dotted lines 80a-80c in FIG. 4.

The number of functional pipeline stages 60a-60m is equal to the number of the programming engines 16a and 16b in the functional pipeline units 50 and 70. This ensures that a particular pipeline stage executes in only one programming engine 16 at any one time.

Referring to FIG. 5, functional pipeline units 50, 70, and 90 are shown to include the programming engines 16a (PE0), 16b (PE1), and 16c (PE2), respectively, in addition to the data units 52a-52c. Between the programming engines 16a-16c, critical

sections 82a-82c and 84a-84c are provided. The critical sections 82a-82c and 84a-84c are used in conjunction with shared data 86a-86c and 88a-88c.

In the critical sections 82a-82c and 84a-84c, the programming engine contexts 58a-58c are given exclusive access to the shared data 86a-86c (e.g., cyclic redundancy check residue (CRC), reassembly context, or a statistic) in external memory.

In operation, functions can be distributed across one or more functional pipeline stages 60a-60d. For example, the critical section 82a represents a section of code where only one programming engine context, in this case, the context 58a of the programming engine 16a, has exclusive modification privileges for a global resource (i.e., shared data 86a), such as a location in memory, at any one time. Thus, the critical section 82a provides exclusive modification privileges to a particular functional pipeline stage of the programming engine 16a. The critical section 82a also provides support for exclusive access between the contexts 58a-58c in the programming engines 16a-16c.

In certain implementations only one function modifies the shared data 86a-86c between the programming engines 16a-16c, to ensure exclusive modification privileges between the programming engines 16a-16c. The function that modifies the shared data 86a-86c executes, e.g., in a single functional pipeline stage 60a,

and the functional pipeline unit 50 is designed so that only one programming engine from all the programming engines 16a-16c executes the functional pipeline stage 60a at any one time.

Still referring to FIG. 5, each of the programming engines 16a-16c is assigned exclusive modification privileges to the non-shared data 86a-86c and 88a-88c, satisfying the requirement that only one function modifies the non-shared data 86a-86c and 88a-88c between the programming engines 16a-16c.

In this example, by optimizing the control flow through the functional pipeline units 50, 70, and 90, the architectural solution described above presents greater network processing power to the hardware-based multithreaded network processor 12.

In the functional pipeline unit 50, the programming engine 16a transitions into the critical section 82a of the functional pipeline stage 60a unless it can be assured that its "next" programming engine 16b has transitioned out of the critical section 82a. The programming engine 16b uses inter-thread signaling where the signaling states can be used to determine availability of a memory resource or whether a memory resource is due for servicing. There are four ways to signal a next context using inter-thread signaling:

Thread Signaling	Mechanism
1. Signal next thread in the same PE	Local Control and Status Registers (CSR) write
2. Signal a specific thread in the same PE	Local CSR write
3. Signal the thread in the next PE	Local CSR write
4. Signal any thread in an PE	CSR write

Critical sections such as CRC calculations are performed in the order of incoming data packets 54a-54c because inter-thread signaling is performed in order.

When the functional pipeline stage 60a transition occurs between the programming engines 16a and 16b, the elasticity buffer 92, implemented as a ring, is used. Each functional pipeline stage 60a is implemented to execute within the time allocated to the functional pipeline stage 60a. However, the elasticity buffer 92 accommodates jitter in the execution of the functional pipeline unit 50. Thus, if a functional pipeline stage 60a falls behind in execution due to system anomalies such as high utilization of memory units over a short period of time, the elasticity buffer 92 allows the context 58a of the functional pipeline stage 60a to be buffered so that the previous functional pipeline stages will not be stalled waiting for the next pipeline stage to complete. The elasticity buffer 92 as shown in FIG. 5 also allows different heartbeats to the different functional

pipelines units 70 and 90.

By using the functional pipeline stages, as described above, the functional pipeline units 50, 70, and 90 cover memory latency and provides sufficient computation cycles for data packets 54 arriving faster than a single stream computation stage. By providing a mechanism for fast synchronization from one programming engine to the next programming engine which performs the same set of functions on a new set of data packets 54, parallel processing capability of the programming engines are greatly enhanced. Multiple means for passing functional state and control is thus provided.

Other Embodiments:

In the example described above in conjunction with FIGS. 1-5, the computer processing system 10 may implement programming engines 16 using a family of network processors, namely, Intel Network Processor Family chips designed by Intel® Corporation, of Santa Clara, CA.

It is to be understood that while the invention has been described in conjunction with the detailed description thereof, the foregoing description is intended to illustrate and not limit the scope of the invention, which is defined by the scope of the appended claims. Other aspects, advantages, and modifications

are within the scope of the following claims.

09/25/2006 15:04:25